CLEAN: A Race Detector with Cleaner Semantics

Cedomir Segulja and Tarek S. Abdelrahman
The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
{seguljac, tsa}@eecg.toronto.edu

Abstract

Data races make parallel programs hard to understand. Precise race detection that stops an execution on first occurrence of a race addresses this problem, but it comes with significant overhead. In this work, we exploit the insight that precisely detecting only write-after-write (WAW) and read-after-write (RAW) races suffices to provide cleaner semantics for racy programs. We demonstrate that stopping an execution only when these races occur ensures that synchronization-free-regions appear to be executed in isolation and that their writes appear atomic. Additionally, the undetected racy executions can be given certain deterministic guarantees with efficient mechanisms.

We present CLEAN*, a system that precisely detects WAW and RAW races and deterministically orders synchronization. We demonstrate that the combination of these two relatively inexpensive mechanisms provides cleaner semantics for racy programs. We evaluate both software-only and hardware-supported* CLEAN*. The software-only* CLEAN *runs all Pthread benchmarks from the SPLASH-2 and PARSEC suites with an average 7.8x slowdown. The overhead of precise WAW and RAW detection (5.8x) constitutes the majority of this slowdown. Simple hardware extensions reduce the slowdown of* CLEAN*'s race detection to on average 10.4% and never more than 46.7%.*

1. Introduction

Data races are one of the key reasons why today's parallel programming is hard. Races cause programs to execute incorrectly [9]. Further, races can give rise to nondeterminism, complicating the entire parallel program development cycle [17]. Finally, and most seriously, the presence of races makes reasoning about possible program executions extremely difficult – previous attempts to define simple execution semantics for racy programs have failed [3, 10]. This inability leads many in both academia and industry to regard *all data races as errors* [20, 13, 2, 26, 27, 11].

Data-race-free programs may still contain bugs. However, they are free of bugs caused by races, which are particularly difficult to reason about [2]. Further, data-race-free programs can be executed deterministically with very little added cost [53]. This makes *race-freedom* a desirable property.

Existing approaches to guaranteeing race-freedom, both *static* and *dynamic*, are impractical. Static approaches identify races at compile time, allowing them to be treated as compilation errors (e.g., [21]), or require using new languages that by design prevent races (e.g. [8]). The former are constrained by decidability limitations and lead to missing or falsely reporting races. The latter are promising but currently restrictive and not widely used [1].

In contrast, dynamic approaches throw an exception during run-time when a first race occurs, (e.g., [20]), thus allowing only a race-free prefix of a program to execute. They can precisely identify all races that occur in the execution being monitored, but even with considerable hardware support they can introduce up to 3x slowdown [19].

This work exploits the insight that many of the difficulties caused by races can be prevented by precisely detecting only certain types of races. Specifically, stopping executions only when a write-after-write (WAW) or a read-after-write (RAW) race occurs suffices to ensure that (i) data within a synchronization-free-region (e.g., a critical section or unprotected code *between* two critical sections) never changes due to a concurrent write, and (ii) writes of synchronization-free-regions appear atomic. Additionally, once the only remaining races are write-after-read (WAR) ones, efficient mechanisms of weakly deterministic systems [45] suffice to to provide deterministic results in *racy* executions that are not stopped.

A system that stops an execution only on WAW and RAW races and deterministically orders synchronization operations gives rise to a novel execution model that has several desirable properties. First, it provides cleaner semantics for racy programs since executions are guaranteed to produce values that necessarily exist in program code, i.e., there are no "out of thin air" values [36]. Second, the produced values are always the same for a given input, i.e., they are deterministic. Finally, this execution model lends itself to an efficient implementation since the type of race that is most expensive to detect (WAR) no longer needs to be detected.

Based on these observations, we designed a system called CLEAN. It precisely detects all WAW and RAW races, and deterministically resolves synchronization operations. We de-

ISCA'15, June 13-17, 2015, Portland, OR USA
 ISBN 978-1-4503-3402-0/15/06...$15.00
DOI: http://dx.doi.org/10.1145/2749469.2750395

sign and evaluate both software-only and hardware-supported implementation of CLEAN. The software-only implementation runs all Pthread benchmarks from the SPLASH-2 [56] and PARSEC [7] suites with an average 7.8x slowdown, making it possibly fast enough to use during code development. Most of this slowdown (5.8x) is caused by intercepting every potentially shared memory access and testing it for races. Hence, we design and evaluate architectural support for precisely detecting WAW and RAW races. We demonstrate that simple hardware additions bring the slowdown caused by WAW and RAW race detection to only 10.4% on average.

This paper makes the following contributions:

- it introduces an execution model that provides cleaner semantics for racy programs,
- it shows that this execution model can be efficiently realized using inexpensive mechanisms, and
- it presents an evaluation of both software-only and hardware-supported implementations of this model.

The rest of this paper is organized as follows. Section 2 discusses background. Section 3 defines CLEAN's execution model and its requirements. Section 4 describes software-only CLEAN, while Section 5 describes CLEAN hardware support. Section 6 presents our evaluation. Section 7 discusses related work, and Section 8 concludes.

2. Background

2.1. Data Race

Two memory accesses constitute a *data race* if they access the same address, at least one is a write, and neither *happens-before* the other. The happens-before relation [28] establishes a partial order over executed accesses. An access a happens-before b if (i) a and b are executed by the same thread and a precedes b in the program order, (ii) a and b are executed by different threads T_a and T_b, respectively, and the execution of a was followed by synchronizing T_b with T_a (e.g., by T_b acquiring the lock released by T_a), only after which b happens, or (iii) there is some access c and a happens-before c and c happens-before b. We distinguish the three classical types of races: a *read-after-write (RAW)*, a *write-after-write (WAW)*, and a *write-after-read (WAR)* race [23].

2.2. Synchronization-Free Regions

A *synchronization-free region (SFR)* is a region of code that starts and ends with a synchronization operation (including thread start/end) and contains no intervening synchronization (e.g., a critical section or code between two critical sections).

An SFR is executed in *isolation* if a data change within an SFR is only ever caused by that SFR alone. An SFR is *write-atomic* if its writes always appear to other threads as executed in a single step[1].

The lack of SFR isolation and write-atomicity can lead to surprising program behaviours where a program produces a seemingly impossible, *out of thin air* [36] value that never appears in the program code. Without SFR isolation, compiler optimizations can expose surprising results to programmers, even leading to security risks. An example of this behaviour, taken from [10], is shown in Figure 1a. Without SFR write-atomicity, programmers can be exposed to similarly unexpected results where, for example, two concurrent SFRs produce data composed of half of the writes of one SFR and half of the other [11], like in Figure 1b.

With today's multiprocessors, SFR isolation and write-atomicity are guaranteed for *race-free* programs [2]. For these, compilers can safely assume that data within an SFR cannot change due to a concurrent action and can thus safely perform optimizations they commonly perform on sequential code. Further, programmers do not have to reason about the granularity of machine instructions since either all or no writes of an SFR appear to be executed.

Regrettably, SFR isolation and write-atomicity are *not* guaranteed for racy code. Thus, in the presence of races, C/C++ compiler optimizations *can* lead to out of thin air values [26, 27]. The previously described scenario of "half-half" data *can* materialize. When this happens, programmers have to understand compiler and hardware intricacies in order to recognize that a race is at the root of the observed behaviour.

For managed languages like Java, where out of thin air values must be disallowed, the lack of hardware (or runtime) enforced SFR isolation requires preventing compilers from making certain optimizations. Specifying allowed optimizations in a way that prevents out of thin air values without completely outlawing common compiler optimizations is an extremely difficult task. It led to a buggy Java memory model [3] that is hard to understand for both programmers and compiler writers.

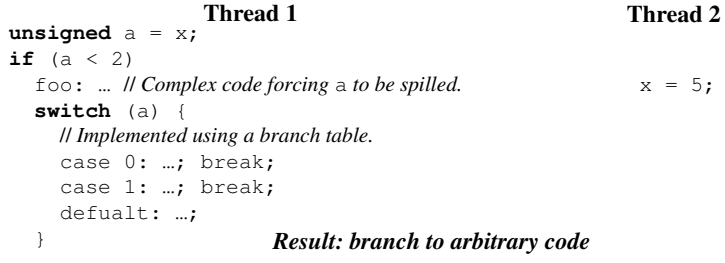
SFR isolation and write-atomicity alone do not address all the difficulties of parallel programming. For example, they do not guarantee that a parallel execution can be seen as a serialized execution where only one SFR is executed at a time [33], and they do not guarantee deterministic execution [16]. However, having SFR isolation and write-atomicity for both race-free and racy codes would provide significantly cleaner semantics: *programmers never observe values that synchronization-delimited parts of the code cannot produce when examined in isolation*.

2.3. Precise Dynamic Data Race Detection

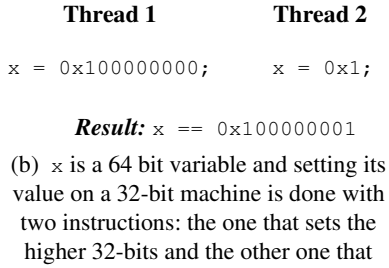
Vector Clocks. A vector clock [22, 39] is a vector of integers that represent program progress, one integer for each thread. Vector clocks are used at run-time to track the happens-before relation, and hence detect races. During execution, one vector clock is maintained for each thread, for each lock, and two for each memory location (one for read and one for write accesses). We refer to the element of a thread's vector clock that corresponds to the thread itself as the *main element*.

The algorithm for maintaining vector clocks ensures the following. A previously executed read/write to location x

[1] In the database community, these two properties are provided for transactions by snapshot isolation [4].



(a) Compiler optimizes Thread 1 code unaware of the racy write by Thread 2. First, when `a` is spilled its value is not saved; rather `a` is reloaded from `x`. Second, the bounds checks and the default case of the switch statement are removed, since `a` (and `x`) seem to be either 0 or 1. Racy write causes the branch table to be accessed with an out-of-bounds index of 5.



(b) `x` is a 64 bit variable and setting its value on a 32-bit machine is done with two instructions: the one that sets the higher 32-bits and the other one that sets the lower 32-bits. The lack of write-atomicity produces a nonintuitive result where `x` has a value that never appears in the code.

Figure 1: The lack of SFR isolation and write-atomicity leads to seemingly impossible executions. `a` is private, `x` is shared.

happens-before a current access to x if and only if every element of the read/write vector clock of x is smaller than or equal to its counterpart in the current thread's vector clock.

This element-wise comparison is made on each access. A race occurs if and only if a thread reads/writes location whose write (or in case of a write either read or write) vector clock element is greater than the counterpart element of the thread's vector clock. After this check is performed, the thread sets the read/write (based on the access type) vector clock of the accessed location to the current value of its vector clock.

Vector clocks of threads and locks are updated less frequently, on thread creation, termination, and synchronization; the reader is referred to [48] for the exact algorithm.

FastTrack. FastTrack [23] is a precise race detector that improves on the basic scheme above. Its insight is that to detect WAW and RAW races it suffices to check if the current access to a location races only with the *last* executed write to that location. This eliminates the need to maintain write vector clocks (which track the last write of each thread).

When a thread executes a write, it suffices to save only the thread's identifier and the main element of the thread's vector clock. This pair can be represented with a single integer – *epoch* – with higher bits of the epoch dedicated to the thread id and lower to the (scalar) clock. When a thread accesses memory, a WAW or a RAW race occurs if and only if the clock component of the saved epoch is greater than the current thread's vector clock element corresponding to the thread indicated by the id component of the epoch [23].

Epochs reduce the space overhead and replace n clock comparisons (where n is the number of threads) done on each access, with only a single comparison. Unfortunately, a similar optimization cannot be done for WAR races: a write *can* race with some previously executed read to the same location without racing with the last executed read to the same location. Thus, in order not to miss a WAR race, it is (still) necessary to maintain a complete read vector clock for some locations, and to do n clock comparisons when accessing these locations.

2.4. Deterministic Synchronization

Deterministic synchronization ensures that threads execute synchronization operations in some arbitrary, but always the same order. In the absence of data races, a deterministic synchronization order guarantees a deterministic program result[2]. Several synchronization orders and algorithms for generating them have been proposed [6, 16, 31, 45]; we use Kendo [45].

In Kendo, each thread maintains a counter that is incremented on executing deterministic events (e.g., on each instruction). A thread can execute a synchronization operation if and only if the value of its deterministic counter is smaller than the counters of all other threads (thread id is used to break ties). Otherwise, the thread must wait for other threads to "catch up", increasing their deterministic counters and eventually guaranteeing progress of the waiting thread.

3. CLEAN

We start by stating the guarantees that CLEAN gives for executions of parallel programs. Then we describe two mechanisms used to provide these guarantees, state their conceptual requirements, and discuss their correctness.

3.1. CLEAN Execution Model

3.1.1. Properties. CLEAN executions have the following properties:

- a *race exception* is thrown if and only if a WAW or a RAW race occurs, at which point the execution stops,
- SFR isolation and write-atomicity are guaranteed for all executions,
- exception-free executions are deterministic.

These properties are achieved by combining precise WAW and RAW race detection with deterministic synchronization.

SFR isolation requires that a thread executing an SFR never observe data change due to concurrently executing code. Such an observation necessarily implies a RAW race: the thread executing the SFR must be reading a value written by a racy write (e.g., in Figure 1a the write to `x` by Thread 2 races with the read of `x` in the optimized `switch (a)` statement). Thus, raising exceptions on RAW races suffices to detect all potential SFR isolation violations.

SFR write-atomicity requires writes of an SFR to appear as executed in a single step. This is true when (i) reads observe

[2]There are sources of nondeterminism other than thread communication but their impact is smaller [53]; we do not address them in this paper.

writes of an SFR only after this SFR finishes, and (*ii*) SFRs that write to same data never execute concurrently (otherwise their writes can appear interleaved as in Figure 1b). The first condition is met when there are no RAW races: reading a value written by a not-yet-finished SFR is necessarily a RAW race. The second condition is met when there are no WAW races. Thus, raising exceptions on RAW and WAW races suffices to detect all potential SFR write-atomicity violations.

Once only executions with no WAW or RAW races are allowed to complete, deterministic synchronization suffices to ensure that completed executions give deterministic results (even when WAR races occur). Since each SFR executes in isolation, synchronization points are reached deterministically. By induction then, deterministically resolving only synchronization operations suffices to make the entire execution deterministic.

CLEAN can raise a race exception in one program execution (in which case a race must have occurred) but can allow a different program execution to complete (in which case the result is guaranteed to be deterministic). The choice between an exception and exception-free run depends on the timing of a read racing with a write. If this race resolves as a RAW, a race exception is thrown. Otherwise, the execution continues.

3.1.2. Benefits. We summarize the benefits of the CLEAN execution model below.

Early Detection of Races. A silently executed race can lead to catastrophic consequences (e.g., [30]). CLEAN detects WAW and RAW races and immediately stops execution, signaling a programming error during testing and preventing further damage during production.

Cleaner Semantics for Racy Programs. CLEAN guarantees that SFRs in all executions appear to be executed in isolation and that they are write-atomic. As discussed in Section 2.2, the lack of these guarantees exposes programmers to surprising values that none of the synchronization-delimited parts of the code, examined in isolation, can produce. In order to fix these errors, programmers have to understand the details of compiler and hardware operations. CLEAN ensures that programmer never observe out of thin air values. It also allows compilers to freely perform optimizations[3].

Determinism. CLEAN ensures that all executions of race-free programs are deterministic. This improves programmability of race-free-programs the same way *weakly* deterministic systems do [45]. Thus, deterministic programs can be more easily tested and debugged, and can be used in domains that require repeatable results (e.g., CAD tools [34], multi-threaded replicas [57]).

Although CLEAN does not guarantee reproducibility for racy programs, it still simplifies testing and debugging. First, since a program is allowed to produce only one correct result, it is easier to detect incorrect behaviours. For example, multithreaded replica-based fault tolerance systems that reach a correct consensus using a quorum can now more easily distinguish between the "correct" (all that finish) and "incorrect" (all that cause exceptions) executions.

Second, if a program execution completes without a race exception but with an observably incorrect result, all subsequent executions will necessarily discover program errors, since the executions will either end with the exactly same incorrect result or with a race exception. Finally, if a program execution does trigger a race exception, a precise race detector can be used alongside CLEAN in subsequent runs to systematically detect all races.

[3]Compilers must still not introduce races in race-free programs [10]. This would lead to exceptions in, from programmers' perspective, race-free code.

3.2. WAW and RAW Race Detection

Basic Mechanism. To detect WAW and RAW races, CLEAN simplifies the FastTrack algorithm (Section 2.3). It maintains the following metadata. First, it maintains a vector clock for each running thread and for each lock. These are updated on synchronization and thread create/join operations as in standard race detectors based on vector clocks. Second, it maintains one epoch for each shared memory location (these can be conservatively identified by a compiler, Section 4).

On each shared memory access, CLEAN executes the algorithm in Figure 2. The `EPOCH_ADDRESS` macro returns the address of the epoch of a given memory location. `CLOCK` and `TID` extract the corresponding epoch components, while `EPOCH` creates a new epoch for a given thread id and clock.

```
    typedef struct {
      int tid;    // Thread's identifier.
      int vc[];   // Thread's vector clock, one element
                  // for each thread
    } thread;

    void checkAccess(void *addr, bool read,
                     thread *t) {
      // Load the epoch of this memory location.
1:    int* epochAddr = EPOCH_ADDRESS(addr);
2:    int epoch = *epochAddr;

      // Check for WAW or RAW race.
3:    if (CLOCK(epoch) > t->vc[TID(epoch)]) error;

      // If write, update the epoch of addr if needed.
4:    int newEpoch = EPOCH(t->tid, t->vc[t->tid]);
5:    if (!read && epoch != newEpoch)
6:      *epochAddr = newEpoch;
    }
```

Figure 2: Data race check in CLEAN.

The epoch of a memory location, updated on line 6, holds the information about the last write to that location: the id of the last thread to perform the write, and the main element of the thread's vector clock at the time of the write. To see if a memory access constitutes a WAW or a RAW race, CLEAN compares the clock component of the saved epoch with the current thread's vector clock element corresponding to the thread that last wrote to the location in question (line 3).

For correctness, CLEAN must maintain epochs and execute checks for races at the finest granularity at which a program may access memory, i.e., for each byte. While a specialization for type-safe languages would allow for a coarser granularity

(e.g., the object granularity), we do not explore it since the scope of such implementation would be limited.

Atomicity. When race checks for accesses to the same data are being performed concurrently, care is needed not to miss or falsely report a race, and to ensure atomic update of metadata. In the case of CLEAN, concurrently occurring checks for write accesses can lead to missing a WAW race when concurrent checks first load an epoch at line 2 in Figure 2, only then to update the epoch at line 6. Further, a RAW access can be misinterpreted as a WAR access. This occurs when a write access and its check are interleaved with a concurrent read access and its check in such a way that the read and its check happen after the write but before the write's check. This would lead to missing a RAW race, which must be detected.

In general, the correctness of race detection can be ensured by using locks to prevent checks to overlap in time. However, this approach leads to a significant overhead – more than 40% of the total detection overhead has been attributed to locking [19, 38]. In the case of CLEAN, both issues described above can be resolved without the overhead of locking on each memory access. This is achieved by different means for software-only implementation (Section 4.3), and when dedicated hardware support is available (Section 5.1).

3.3. Deterministic Synchronization

CLEAN uses the Kendo algorithm (Section 2.4) to deterministically order synchronization operations. This requires CLEAN to maintain one counter per each running thread, and to increment this counter on deterministic events. This functionality can be achieved using hardware performance counters [45], or through compiler instrumentation [5].

Further, synchronization operations must be modified so that a thread performs synchronization only when its deterministic counter is the minimum across all counters, or waits otherwise. The thread creation routine must be modified in the same way to ensure that thread ids are deterministic [45].

3.4. Correctness

To prove that the described mechanisms indeed implements the execution model described in Section 3.1, we need to show that (*i*) a race exception is raised if and only if a WAW or a RAW race occurs, (*ii*) SFR isolation and write-atomicity are maintained in all executions, and (*iii*) exception-free executions are deterministic. The first claim is evident: CLEAN detects, and raises an exception, for all races detected by a precise detector but WAR races.

To prove the last two claims, we first show that with CLEAN, the following proposition holds: *a read always returns the data written by the immediately preceding happens-before write*. First, since CLEAN stops an execution on a WAW race, all writes to same data that are allowed to execute are necessarily ordered by happens-before. Thus, "the immediately preceding happens-before write" is well-defined, since there is exactly one such write. Second, since CLEAN stops an execution on a RAW race, a read that is allowed to execute necessarily reads a data written by the immediately preceding happens-before write.

There are three consequences of the above proposition. The first is SFR isolation. A read within an SFR cannot observe a value of a write performed concurrently with the SFR, since that write does not happens-before the read.

The second consequence is SFR write-atomicity. Observing writes of an SFR only if those writes (and hence the entire SFR) happens-before implies that *all* writes of the writing SFR propagate to the thread executing the reading SFR before *any* can be observed. And writes of different SFRs never appear interleaved since writes to the same data, and hence the SFRs executing them, are ordered by happens-before.

Finally, when synchronization is deterministic, determinism of exception-free runs also follows from a read returning the data written by the immediately preceding happens-before write. CLEAN uses Kendo to ensure that happens-before relation is deterministic; hence, the immediately preceding happens-before write is the same in all executions.

4. Software-only CLEAN

CLEAN can be implemented completely in software. Deterministic synchronization has been previously efficiently implemented in software; see [45] for details. Here we focus on a software-only WAW and RAW race detection.

4.1. Compiler Instrumentation and Run-Time

A conservative estimate of shared accesses regards all accesses as shared except those to local variables that never have their address taken. This information is provided by modern compilers with no additional work, e.g., LLVM [29] lowers stack-allocated scalars that do not have their address taken to virtual registers. Escape analysis (e.g., [5]) can be performed to improve this estimation, but we do not explore it.

A call to a run-time routine that implements a race check is inserted for each identified shared access. Compared to Figure 2, the race check implements one optimization. Similar to FastTrack [23], we extend vector clock elements to epochs by setting the highest bits of each element to the corresponding thread identifier. Although these bits are redundant (the thread id can be determined by the position of the element within the vector clock array), they allow for a direct comparison of epochs and vector elements at lines 3 and 5 in Figure 2.

4.2. Epoch Organization

We represent epochs with 32-bit integers and save them in a dedicated region of the process address space. Since CLEAN never requires inflation of epochs to vector clocks, this region has a fixed layout, allowing for an efficient `EPOCH_ADDRESS` macro. There is a one 4 byte epoch for each byte of program data and it is placed at the address $epochs\_base\_address + 4x$, where x is the address of the data, and $epochs\_base\_address$ is chosen to avoid overlaps with program data.

This organization requires a large, fixed size region in the process virtual address space. However, only the epochs for

the accessed program data ever need to be allocated physical memory. Hence, the CLEAN memory requirements are proportional to the size of the accessed shared program data.

4.3. Atomicity

To ensure that a WAW race is correctly detected in the case of concurrent write checks, the update of the epoch in Figure 2 at line 6 is replaced by a compare-and-swap (CAS) instruction that atomically compares the current value of the epoch with the previously loaded one, and only performs the update if these two values match. Otherwise, a WAW race occurred and a race exception is raised.

Mistaking a RAW for a WAR race is avoided by performing the check on a write access *before* the actual write, while the check on a read is performed immediately *after* the read. On modern x86 processors, which allow only for later loads to be reordered with earlier stores [25], guaranteeing that checks occur before writes and immediately after reads requires no memory barriers.

These two extension ensure precise detection of all WAW and RAW races. We omit the proof for space considerations.

4.4. Multi-Byte Accesses

As discussed in Section 3.2, CLEAN must check for races at byte-granularity. This presents a performance challenge since, for example, 8 checks would have to be made on single 8-byte access, resulting in a significant slowdown. Here we discuss how to decrease the latency of checks on multi-byte accesses.

First, we note that if an epoch update is needed, the epoch of all bytes involved in a multi-byte access will be updated to the same value. Since epochs of adjacent bytes are adjacent in the memory, we can use a wide CAS instruction to update multiple epochs simultaneously (most today's 64-bit processor have a 128-bit wide CAS, which allows for simultaneous update of 4 epochs).

Second, we observe that for most multi-byte accesses the epochs of all the bytes involved in the access have the same value. In this common case, it suffices to test the epoch of only one of the bytes involved with the access (there is a race on either all or none bytes). Similarly, when comparing an epoch to see if an update is required, it also suffices to compare with only one epoch (but all need to be updated).

Vector instructions can be used to quickly verify if the epochs of all the bytes involved in the access are the same. In the common case, a vector load is used to load multiple epochs simultaneously (e.g., Intel AVX instructions allow for simultaneous load of 256 bits, or 8 epochs), after which a vector comparison establishes equality of all epochs.

4.5. Clock Rollover

CLEAN uses a fixed-size, 32-bit epoch representation. Hence, both the clock and the thread id component are limited.

A thread id can be safely reused once the thread is joined. Thus, although a fixed-size epoch limits the number of concurrently running threads, allocating an adequate number of bits to the thread id is probably sufficient for most practical uses (e.g., 8 bits for up to 256 concurrently running threads).

The limited number of bits in the clock component of an epoch can seriously limit the number of synchronization operations in a program because these clocks are incremented on synchronization operations. If a clock is assigned a value greater than its representation allows for, i.e., after it "rollovers", the CLEAN guarantees could be compromised.

We avoid this correctness issue, albeit at potential performance cost. When a rollover is about to happen, an execution is brought to a halt at the next globally deterministic execution point – when all threads are trying to execute a synchronization operation (or have finished). At this point, all epochs and vector clocks are reset, and the execution resumes.

Resetting all the metadata can lead to CLEAN missing a race in which one access involved in the race was performed before the reset point and the other access happens after it, since the record of the earlier access has been lost. However, SFR isolation, write-atomicity, and determinism *are* preserved. An execution in which resets occur can be observed in phases that end on a reset. Since resets occur at deterministic points and only at SFR boundaries, per-phase SFR isolation, write-atomicity and determinism ensure that the same guarantees are maintained for the entire execution.

On reset, instead of zeroing the entire (large) memory region dedicated to epochs, we simply change the page mapping of the corresponding pages to point to the OS-provided, copied-on-write, zero-page [12]. As Section 6 demonstrates, resets are infrequent and the cost of waiting for a program to reach the next globally deterministic execution point, and resetting the metadata, is negligible in practice.

4.6. Summary of Overheads

Software-only CLEAN can lead to an increase in the execution time due to: (i) intercepting each potentially shared access via instrumentation, (ii) the latency of race checks, (iii) increased memory pressure due to metadata, (iv) altering synchronization operations to maintain vector clocks and achieve deterministic synchronization, and finally (v) occasional deterministic resets of metadata to prevent clock rollovers. Out of these, the time added on each shared access, (i) and (ii), plays the most significant role. The vectorization optimization targets (ii) and we expect it to bring noticeable gains.

CLEAN increases the memory footprint of a program by requiring 4 bytes for each accessed shared program byte. This is strictly smaller than memory requirements of a precise race detector such as FastTrack [23].

5. Hardware support for CLEAN

The race detection of CLEAN is amenable for efficient hardware implementation, mainly because of the fixed memory layout of epochs and the simplicity of race checks. This offers an opportunity to alleviate the overhead associated with intercepting each potentially shared memory access – a significant contributor to the overhead of the software implementation.

5.1. Basics

The hardware support for the CLEAN data race detection (Figure 3) consists of the logic that implements the data race check from Figure 2, one additional 32-bit register, and of augmented cache-coherence messages.

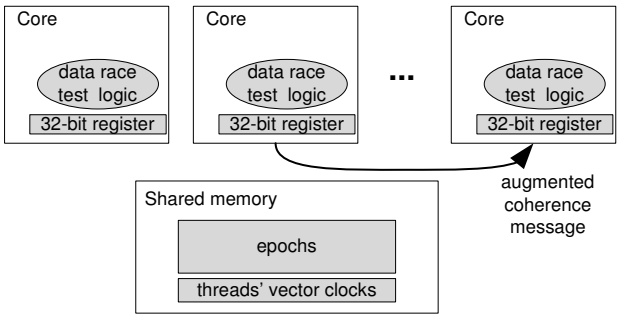


Figure 3: Hardware support for CLEAN in the conventional multiprocessor architecture.

The epochs are represented with 32-bit integers and stored in shared memory, requiring no dedicated storage. They are placed in a fixed area of the process address space so the hardware circuitry can easily access them, as described below. In parallel with each potentially shared access, CLEAN hardware loads the epochs of data being accessed, and performs the WAW and RAW race check.

The race check also requires access to the vector clock of the currently running thread (lines 3 and 4 in Figure 2). To avoid performing an additional memory access, we maintain a per-core copy of the vector clock of the currently running thread. However, not all the elements of the vector clock need to be accessed on each race check. In fact, the common case only requires access to the main element of the vector clock, and consequently we only dedicate storage to this 32-bit integer. The value of this register is changed only on performing (i) a synchronization operation and (ii) a context switch. CLEAN makes this register visible to software to allow for the update upon executing these events.

To ensure atomicity during WAW race checks and proper identification of RAW races, we use the insight that any concurrent remote racy accesses will necessarily generate cache coherence traffic [19]. Thus, if while performing a data accesses and the corresponding race check, a processor receives a coherence message regarding the accessed data, CLEAN hardware reports a data race. In order not to falsely identify concurrent accesses to disjoint parts of a cache line as a data race, CLEAN requires cache coherence messages to include, in addition to the cache line identifier, the position of the accessed data within the requested cache line.

5.2. Data Race Check Logic

The hardware performs a race check in parallel with an access to each potentially shared byte in memory (Figure 4a). The data race check is equivalent to the software-only one, with one optimization. The check for data race is unnecessary when the thread currently performing the access and the thread that previously wrote this data are the same – no data race can occur in this case (*sameThread* check in the figure).

This observation, together with the fact that no epoch update is necessary on read access or when the epoch is already up to date (*sameEpoch* check), allows us to swiftly complete the race check in certain cases. Once the epoch of the data byte being accessed is loaded, the simple circuitry in Figure 4b compares the epoch with the cached main element of the thread's vector clock and signals if no further work (discussed below) is needed. Our evaluation (Section 6) shows that the majority of accesses can be resolved in this way.

When the *sameThread* resolves as false, the hardware loads the needed element of thread's vector clock, compares it to the epoch, and raises an exception if a race has occurred. In case of a write and the *sameEpoch* resolved as false, the hardware updates the value of the epoch in the memory before continuing program execution.

To efficiently handle multi-byte accesses, the described circuitry is replicated so race check for all accessed bytes can be performed concurrently. Multiple epochs can be loaded, tested, and updated in parallel, as long as they belong to the same cache line. Otherwise, the race check is performed in multiple sequential steps, one step for each epoch cache line.

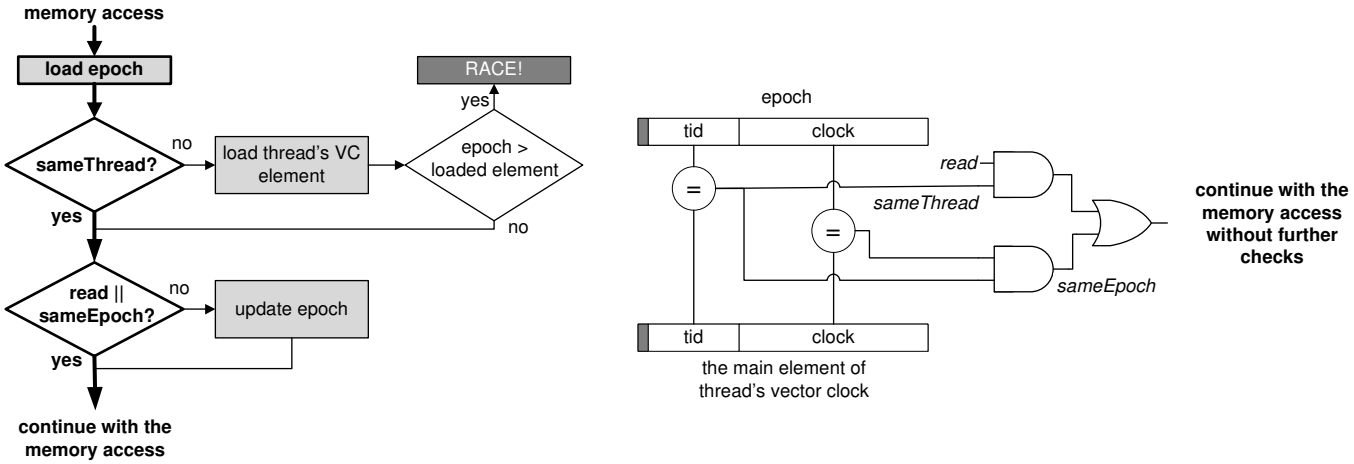
5.3. Metadata Organization

Epochs and threads vector clocks are placed in a fixed area of the process address space (Figure 5a) and are laid out in a predetermined manner so that hardware can easily calculate the addresses of the epoch and the element of the vector clock that are required during data race check[4].

Epochs for adjacent data are placed adjacent in memory. Since epochs take advantage of the memory hierarchy like regular data, this layout allows exploiting temporal and spatial locality of data accesses. However, fetching 4 bytes of metadata on each access to a potentially shared byte of data can cause a significant performance degradation due to increased cache pressure. Further, epochs for data of a single cache line can be spread across multiple cache lines, resulting in a long latency of a data race check.

To avoid this performance degradation, we observe that a more compact, 1:1 metadata organization is possible when 4 consecutive bytes of data all have the same epoch. In this case, we can maintain only one epoch for 4 bytes of data. However, as soon as one of the data bytes in this group of 4 needs to be assigned an epoch different than the rest of the group we need to switch to a 1-epoch-per-data-byte organization to maintain correctness. This occurs, for example, upon a 1-byte write by a thread different than the thread that previously wrote to 4 bytes that include the byte in question. We find that the occurrence of such events in general is very rare (Section 6), and hence optimize for the common case, as described below.

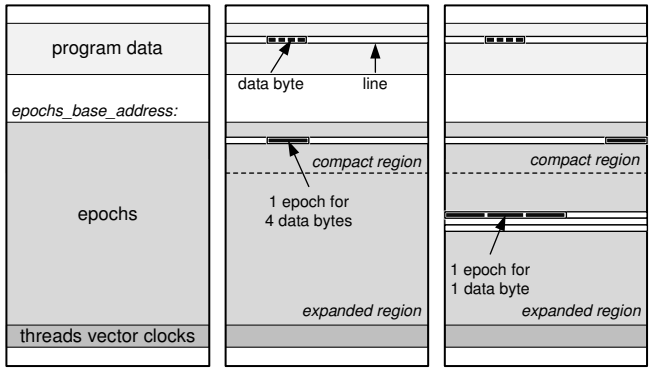
[4]For programs whose data is large enough to overlap with the metadata, additional address bits (unused by current 48-bit implementations of the 64-bit address space) can be used to offset the metadata and prevent the overlap.



(a) Control for race check.The fast (and commonly taken) path is marked bold.

(b) The part of the race check data path resolving the common case. The role of the highest epoch bit (shaded here) is explained in Section 5.3.

Figure 4: Hardware data race check in CLEAN.



(a) Process address space layout

(b) Data to metadata mapping for compact data lines

(c) Data to metadata mapping for expanded data lines

Figure 5: Process address space layout in CLEAN.

The epoch area of the process address space is logically divided into two regions: *compact* and *expanded*, as shown in Figure 5b. We logically organize data and metadata in units of the same size as the cache line and refer to these units simply as *lines*. Each line of data has one corresponding epoch line in the compact region and 3 epoch lines in the expanded region. In each region, epoch lines for adjacent data lines are placed adjacent in memory.

Before any accesses have been made, epochs for a data line are stored in one metadata line: the one that is placed in the compact region. This epoch line holds one epoch for each 4 bytes the data line (Figure 5b). At this point, the data and its epoch line are in the *compact* state.

When a byte in any group of 4 bytes belonging to a data line needs to be assigned a different epoch, the data line is transitioned to the *expanded* state. Its epochs are now spread across 4 epoch lines, first of which takes the space in the compact region previously held by the compact epoch line, as shown in Figure 5c.

To support compact and expanded data lines, the basic data race check logic described before is extended as follows. First, the highest bit of each epoch is used to encode if the data and the corresponding epoch lines are in compact or expanded state[5]. Second, as this information is only known *after* an epoch has been loaded, hardware always assumes the compact state when calculating the address of an epoch. If a loaded epoch signals that the line is in the expanded state, hardware reloads the epoch from now properly calculated address. Note that in the case of incorrect epoch address calculation, the epoch line first accessed can still contain needed metadata, albeit at different offset, as in Figure 5c. Hence, the penalty of epoch address miscalculation is sometimes less than accessing an additional cache line.

Finally, before updating an epoch of a compact line, a test is made to see if a transition to an expanded state is needed. If so, the hardware "stretches" the compact epoch line to 4 expanded epoch lines along with setting the expanded bit of each epoch.

5.4. Summary of Overheads

Hardware-supported WAW and RAW race detection can lead to an increase in the execution time due to: (i) the latency of race checks, (ii) increased memory pressure due to metadata, and to a lesser degree (iii) the latency added by software on synchronization operations to maintain vector clocks.

Since a check for a racy memory access is performed in parallel with the access, its latency is exposed only if longer than the latency of the access. The key factors determining a race check latency are two frequencies: how often is a check completed following the fast path from Figure 4a, and how often is a line that is being checked in a compact state. The latter frequency also impacts memory pressure: checking accesses to compact lines requires accessing less data.

[5]While it suffices to have only one bit per line dedicated to this information, the 1-bit-per-epoch approach allows for more regular epoch layout.

6. Evaluation

6.1. Benchmarks

We use the SPLASH-2 [56] and PARSEC [7] benchmarks. We exclude the `freqmine` benchmark from all our experiments: `freqmine` uses a parallelization API other than Pthread and it is hence not supported by our current implementation. We run the remaining 26 benchmarks with 8 threads, and unless stated otherwise, we use the `native` input, run each experiment 10 times and report the average value and the 95% confidence interval.

To test CLEAN race detection and determinism of exception-free runs, we worked with two versions of the benchmarks: an unmodified, which contains data races in 17 out of 26 benchmarks, and a modified one where we removed all the races reported by ThreadSanitizer [54]. We omit `canneal` from the modified version because it employs a lock-free synchronization strategy which results in too many races to be removed manually. For performance results, we used the modified version (since the unmodified, racy version can result in race exceptions).

6.2. Software-only CLEAN

The machine we use is a 2.2GHz dual-socket 8-core Xeon E5-2660 with 32 GB memory running Linux 3.2.

6.2.1. Implementation. We implemented software CLEAN on top of the ThreadSanitizer [54], a production quality imprecise race detector. ThreadSanitizer consists of a compiler pass and a run-time component. The compiler pass inserts calls to the run-time race check routines before each potentially shared access.

The run-time intercepts Pthread API, maintains vector clocks of threads and locks, and provide mechanisms for maintaining race metadata. It implements a race detection algorithm similar to FastTrack, but trades detection precision for performance. Most notably, ThreadSanitizer can miss a race due to (i) maintaining a record of only k (typically $k = 4$) last access to each 8-byte memory region, and (ii) lack of atomicity for concurrently occurring race checks.

We did the following modification to implement CLEAN. First, we modified the compiler and intercepted Pthread API routines to implement the Kendo algorithm (Section 3.3). We increment deterministic counters on each basic block in which instruction count is greater than some predetermined cutoff value – this reduces instrumentation overhead but can result in our deterministic counters imprecisely reflecting thread progress. Second, we modified metadata organization and race detection routines to implement the race check of CLEAN (Sections 4.1 and 4.2). Third, we modified the compiler pass to properly order run-time calls with respect to read/write memory accesses (Section 4.3). Lastly, we implemented the clock rollover procedure and the multi-byte access optimization (Sections 4.5 and 4.4).

6.2.2. Detected Races and Determinism. It is not possible to prove the correctness of an implementation of the CLEAN execution model empirically (we argue correctness in Section 3.4). Nonetheless, we test that CLEAN throws exceptions only when races occur and that it ensures determinism for exception-free executions, with the following two experiments. First, we run the unmodified benchmarks. We run each benchmark 100 times, and use the `simlarge` input to reduce running time. The executions of all 17 racy benchmarks *always* end with an exception.

We then run the race-free version of the benchmarks, again 100 times and with the `simlarge` input. We collect the program output, the final state of deterministic counters, and the number of shared read and write accesses to verify determinism. The executions of the race-free benchmarks never result in an exception and are always deterministic.

6.2.3. Performance. The execution time under CLEAN normalized w.r.t. the nondeterministic run is shown in Figure 6. An average slowdown of 7.8x makes this implementation too slow for an always-on use, but possibly fast enough to use during development. The figure also shows the performance of CLEAN mechanisms in isolation; the race detection dominates deterministic synchronization in terms of overhead.

Deterministic Synchronization. Most benchmarks experience a small slowdown due to deterministic synchronization. Some actually enjoy speedup: our deterministic synchronization operations use spinning instead of blocking when the number of threads is lower than the number of processors. The impact of this optimization, most notably affecting `streamcluster`, has been previously documented [52].

Some benchmarks do experience significant slowdown, for three reasons. First, we implement deterministic counters using instrumentation; the slowdown of this alone was measured to be up to 36%. Second, a deterministic synchronization operations requires comparing counters of all threads, so it typically has a greater latency than its nondeterministic counterpart. `fmm`, `radiosity`, and `fluidanimate` perform frequent synchronization and reveal this latency. Finally, since our deterministic counters are not incremented on small basic blocks, they can imprecisely represent real thread progress; this affects aforementioned benchmarks as well as `dedup`, `ferret`, and `vips`, whose threads perform significantly different work (e.g., pipeline parallelism).

Instrumentation cost [42], imprecision of counters [45, 5], and the latency of synchronization operations [46] can be lowered. Since the focus of this paper is not optimizing the performance of deterministic synchronization we do not explore these optimizations.

Data Race Detection. A software implementation of race detection requires instrumenting and acting on each potentially shared memory access. This alone results in an average 5.8x and as high as 22x slowdown. As expected, this cost is mostly related to the frequency of shared accesses (Figure 7): the two worst performing benchmarks (`lu_cb` and `lu_ncb`) access shared data significantly more frequently than the other benchmarks.

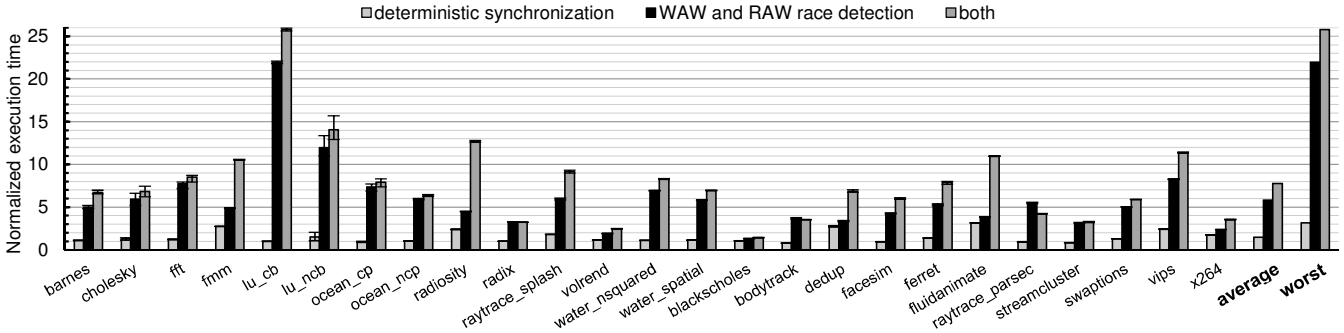


Figure 6: Software-only CLEAN performance. Error bars show 95% confidence interval.

Figure 8 shows the impact of vectorization (Section 4.4). The success of this optimization stems from the following measured values: (i) on average more than 91.9% of shared accesses are 4 or more bytes wide, and (ii) for more than 99.7% of shared accesses in every benchmark, the epochs of all the bytes involved in the access have the same value.

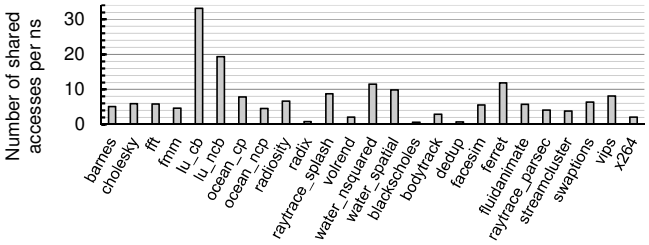


Figure 7: The frequency of shared accesses.

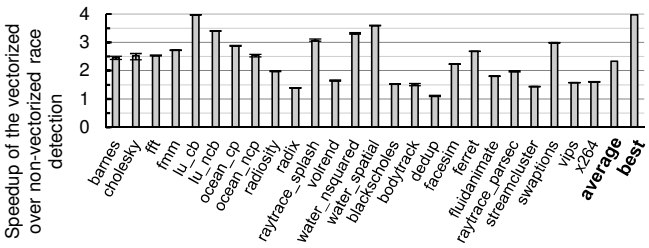


Figure 8: The impact of vectorization.

Clock Rollover. As discussed in Section 4.5, CLEAN prevents clock rollovers, albeit at potential performance cost. In the default configuration, we set the width of the clock component of a 32-bit epoch to 23 bits, leaving 8 bits to reusable thread id (and 1 bit to accommodate for hardware implementation). We then run CLEAN with the clock component set to 28 bits, in which case deterministic reset of metadata is not needed to avoid rollovers in any of our benchmarks. Table 1 shows the number of rollovers with the default configuration normalized w.r.t. the execution time of the nondeterministic run, as well as the decrease in the execution time of the 28-bit configuration relative to the default configuration. The clock rollover impact is very small.

6.3. Hardware supported CLEAN

6.3.1. Simulation. We use simulation to evaluate the performance of the hardware-supported race detection described in Section 5. The simulator is Pin-based [35], and it was previously used to evaluate a hardware-supported implementation of a precise data race detector [19]. It models an 8-core multiprocessor with simple cores (fixed 1 cycle latency for non-memory instructions), but with a realistic memory hierarchy. This hierarchy consists of a private L1 (8-way 64KB), private L2 (8-way 256KB), shared L3 cache (16-way 16MB), all with 64B lines, and a MESI coherence protocol. The memory access latencies are (all in cycles): 1 (L1 hit), 10 (local L2 hit), 15 (remote L2 hit), 35 (L3 hit), and 120 (L3 miss).

Benchmarks	# Rollovers per Second	Execution Time Decrease When No Clock Rollover Occurs
barnes	5.6	2.1%
fmm	4.9	0.8%
radiosity	31.0	1.7%
facesim	8.2	0.0%
fluidanimate	34.8	2.4%

Table 1: The impact of clock rollover. Only the benchmarks experiencing rollovers with the 23-bit configuration are listed.

On each potentially shared memory access (approximated by Pin as non-stack accesses), we faithfully model the race check shown in Figure 4 and its latency by simulating all needed metadata accesses.

We also account for incorrect epoch address calculation (Section 5.3), by adding a penalty of a minimum of 1 cycle and as much as a latency of an additional cache line access. On transitioning a line from its compact to expanded representation, there is a penalty of 1 cycle plus the latency of writing 4 cache lines. Finally, to account for maintaining (software-managed) vector clocks of thread and locks, and the on-chip stored main element of thread's vector clock, we increase the latency of each synchronization operation by 100 cycles.

To reduce simulation time we use the `simsmall` input for all the benchmarks, repeat each experiment 5 times, and omit `facesim` due to too long simulation time.

6.3.2. Performance. Figure 9 shows the execution time when hardware-supported race detection of CLEAN is active normalized w.r.t. the execution in which no race detection is performed (synchronization is not deterministic in either execution). Hardware lowers the penalty of software race detection from on average 5.8x to only 10.4% slowdown.

Figure 10 presents two breakdowns of memory accesses. For each benchmark, the left side of the bar shows the break-

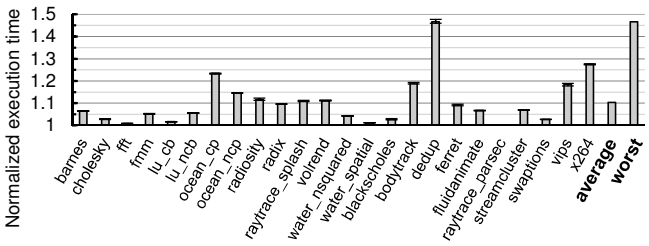


Figure 9: Hardware-supported race detection performance.

down of accesses in terms of complexity of the race check they require (Figure 4a), while the right side of the bar shows how many accesses are made to the lines in compact vs. expanded state. Both of this breakdowns share the *private* category: these represent accesses to private data and do not require any race detection work.

The *fast* accesses follow the fast path from Figure 4a, *VC load* are accesses that require the hardware to load an element of thread's vector clock from the memory and compare it to the previously loaded epoch, *update* are accesses that require epoch update, *VC load & update* accesses require both access to in-memory VC and epoch update, and finally, *expand* accesses are all accesses that trigger a transition of a line from compact to expanded representation. On average 54.2% of accesses are resolved by the fast path. Combined with private accesses, CLEAN hardware quickly checks 90% of all memory accesses. Expensive accesses requiring line expansions were under 0.02% of accesses in every benchmark.

The breakdown of accesses to the lines in compact vs. expanded states shows that most accesses are made to the compact lines. On average, 94.3% of memory accesses either require no metadata to be accessed, i.e., are private, or the size of the accessed metadata is the same as the size of the data. One notable exception is `dedup`, an application that compresses a data stream operating on a single byte granularity. In this case, the majority of accesses are made to the expanded lines, resulting with 46.7% slowdown over baseline.

Due to the simplicity of the hardware race check, where in the common case only the epoch of the data needs to be loaded and quickly compared with the on-chip cached main element of the thread's vector clock, the epoch size is the deciding performance factor. This is clearly shown in Figure 11, which compares the performance of CLEAN hardware-supported race detection, to two alternative designs where the epoch size is 1 and 4 bytes, and no line compaction is applied.

The hypothetical design which assumes that an epoch can fit in just 8 bits presents the upper bound on CLEAN performance. For the majority of benchmarks, CLEAN comes close to this upper bound because of the effectiveness of the line compaction, as seen previously.

On the other hand, the design with 4-byte epochs is representative of the situation where with CLEAN all accesses would be to expanded lines, but without the penalties of line expansion and epoch address miscalculation present. As seen, 4-byte epochs significantly degrade performance, especially for `ocean_cp`, `ocean_ncp`, and `radix`. These benchmarks had the highest baseline LLC miss rate, which additionally significantly rose due to added cache pressure in the presence of metadata accesses to more than 9% in all three benchmarks. CLEAN avoids this performance degradation because of its line compaction.

7. Related Work

Data Race as an Error. Goldilocks [20] was the first work to treat races as exceptions. It detects all races but has high overhead. The worked that followed improved performance by allowing for some races while ensuring that executions are sequentially consistent (SC) [37, 55, 43, 51], or (stronger) region serializable (RS) [33, 47]. These properties also simplify reasoning about racy executions but, in contrast to simple hardware support for CLEAN, guaranteeing them requires spare processors, the use of large dedicated hardware structures, or modifying the cache or the coherence protocol.

CLEAN guarantees determinism in exception-free runs and SFR isolation and write-atomicity in all runs. RS is a stronger property than SFR isolation and write-atomicity. SC is not comparable with SFR isolation and write-atomicity: an SC execution does not provide SFR isolation or write-atomicity and vice versa. RS and SC do not provide determinism.

Dynamic Race Detection. *Precise detectors* [23, 19] detect all races that occur in the execution being monitored, have no false positives, but have high overhead. In contrast, CLEAN focuses on WAW and RAW races and thus maintains smaller and more regular metadata, performs less actions on each accesses (no update of metadata on reads, no check for WAR on writes), and supports atomicity in software while avoiding locking.

RADISH [19] is a precise hardware-accelerated race detector. Both RADISH and CLEAN start from the FastTrack algorithm and their hardware support shares many similarities. Both access metadata in parallel with each shared access, rely on coherence messages for atomicity, and exploit compaction to improve performance. However, there are notable differences. First, RADISH maintains additional in-cache representation of (in-memory) metadata. The size of this in-cache metadata is proportional to the number of processors, and it requires additional actions on coherence messages, cache line evictions and context switches to maintain precision. Second, RADISH performs compaction only of in-cache metadata and relies on type-safety when doing so, which limits its generality. Finally, RADISH occasionally resorts to expensive software race checks. These features let RADISH detect all races, but also lead to up to 3x slowdown.

Imprecise detectors [41, 50, 49, 44, 54] typically have low overhead but can miss races and report non-existing ones. The fundamental difference between them and CLEAN is that their imprecision is due to practical considerations, e.g., limited dedicated hardware resources. CLEAN *chooses* to miss WAR races by design, provides SFR isolation and write-atomicity and ensures determinism when races go undetected, and never results in a false positive. CORD [50] and ReEnact [49] addi-

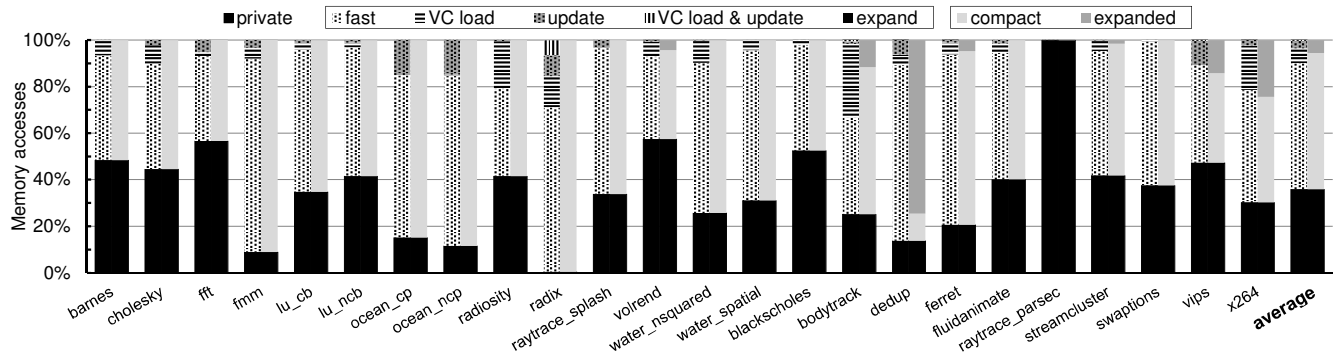


Figure 10: The breakdown of memory accesses.

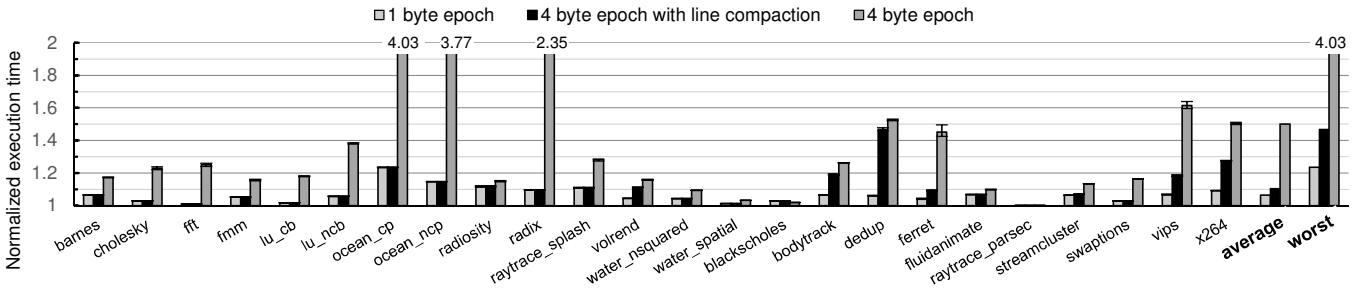


Figure 11: Performance of WAW and RAW race detection with 1 and 4 bytes epoch size.

tionally record the order of synchronization operations and races. This enables deterministic replay, but not deterministic execution.

Deterministic Execution. *Weakly* deterministic systems provide determinism only for race-free codes and do so efficiently in software [45, 15]. CLEAN builds on mechanisms of weakly deterministic systems, in particular of Kendo's, to provide determinism for exception-free executions. However, CLEAN requires hardware support to do so efficiently.

Strongly deterministic systems provide determinism for all programs by silently and deterministically ordering races but have high overhead [5, 31, 40], require complex hardware support [18, 24], or their memory requirements limit their applicability [32]. In contrast, CLEAN provides determinism only for exception-free runs, timely detects WAW and RAW races, and avoids high overhead with only simple hardware support. Further, these systems do not discuss SFR isolation and write-atomicity; [31, 40] seem to provide it, others [5, 18, 24, 32] do not.

Clock Rollover. Accordion clocks [14] present an alternative approach to the clock rollover problem. Accordion clocks reduce the possibility of a clock rollover, but in general do not avoid rollovers and the accompanying correctness issues. Our approach handles clock rollovers without compromising SFR isolation, write-atomicity or determinism of exception-free runs.

8. Conclusions

This paper presents CLEAN, a system that deterministically orders synchronization and stops execution when a WAW or a RAW race appears. The combination of these two mechanisms gives rise to a novel execution model that provides cleaner semantics for parallel programs. Specifically, it ensures that synchronization-free regions appear to be executed in isolation and that their writes appear atomic, along with providing determinism when executions run to completion.

We design and evaluate both software-only and hardware-supported CLEAN. The software-only CLEAN implementation incurs 7.8x slowdown on average, and the majority of that overhead (5.8x) stems from precise WAW and RAW race detection. Through simulation, we show that simple hardware extensions reduce the cost of race detection from 5.8x to 10.4%. With a maximum overhead of 46.7%, this allows for always-on precise WAW and RAW race detection, enabling cleaner semantics for racy programs.

9. Acknowledgements

The authors would like to thank Tianyi David Han, Joseph Garvey and the anonymous reviewers for their insightful feedback. We also thank Joseph Devietti for sharing his RADISH simulator, and Dmitry Vyukov for answering our questions about ThreadSanitizer.

References

[1] "Programming Language Popularity," http://langpop.com, Oct. 2013.

[2] S. V. Adve and H.-J. Boehm, "Memory Models: A Case for Rethinking Parallel Languages and Hardware," *Commun. ACM*, vol. 53, no. 8, pp. 90–101, Aug. 2010. [Online]. Available: http://doi.acm.org/10.1145/1787234.1787255

[3] D. Aspinall and J. Sevcik, "Java Memory Model Examples: Good, bad and ugly," in *Proc. of VAMP*, 2007, pp. 66–80.

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A Critique of ANSI SQL Isolation Levels," in *Proc. of SIGMOD*, 1995, pp. 1–10.

[5] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman, "CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution," in *Proc. of ASPLOS*, 2010, pp. 53–64.

[6] E. D. Berger, T. Yang, T. Liu, and G. Novark, "Grace: Safe Multithreaded Programming for C/C++," in *Proc. of OOPSLA*, 2009, pp. 81–96.

[7] C. Bienia, "Benchmarking Modern Multiprocessors," Ph.D. dissertation, Princeton University, 2011.

[8] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian, "A Type and Effect System for Deterministic Parallel Java," in *Proc. of OOPSLA*, 2009, pp. 97–116. [Online]. Available: http://doi.acm.org/10.1145/1640089.1640097

[9] H.-J. Boehm, "How to Miscompile Programs with "Benign" Data Races," in *Proc. of HotPar*, 2011, pp. 3–3.

[10] H.-J. Boehm and S. V. Adve, "Foundations of the C++ Concurrency Memory Model," in *Proc. of PLDI*, 2008, pp. 68–78.

[11] ——, "You Don't Know Jack About Shared Variables or Memory Models," *Commun. ACM*, vol. 55, no. 2, pp. 48–54, Feb. 2012.

[12] D. Bovet and M. Cesati, *Understanding The Linux Kernel*, 2005.

[13] L. Ceze, J. Devietti, B. Lucia, and S. Qadeer, "A Case for System Support for Concurrency Exceptions," in *Proc. of HotPar*, pp. 6–6.

[14] M. Christiaens and K. D. Bosschere, "Accordion Clocks: Logical Clocks for Data Race Detection," in *Proceedings of EuroPar*, 2001, pp. 494–503.

[15] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant, "Parrot: A Practical Runtime for Deterministic, Stable, and Reliable Threads," in *Proc. of SOSP*, 2013, pp. 388–405.

[16] J. Devietti, B. Lucia, L. Ceze, and M. Oskin, "DMP: Deterministic Shared Memory Multiprocessing," in *Proc. of ASPLOS*, 2009, pp. 85–96.

[17] ——, "DMP: Deterministic Shared-Memory Multiprocessing," *IEEE Micro*, vol. 30, no. 1, pp. 40–49, 2010.

[18] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman, "RCDC: A Relaxed Consistency Deterministic Computer," in *Proc. of ASPLOS*, 2011, pp. 67–78.

[19] J. Devietti, B. P. Wood, K. Strauss, L. Ceze, D. Grossman, and S. Qadeer, "RADISH: Always-on Sound and Complete Ra Detection in Software and Hardware," in *Proc. of ISCA*, 2012, pp. 201–212.

[20] T. Elmas, S. Qadeer, and S. Tasiran, "Goldilocks: A race and transaction-aware java runtime," in *Proc. of PLDI*, 2007, pp. 245–255. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250762

[21] D. Engler and K. Ashcraft, "RacerX: Effective, Static Detection of Race Conditions and Deadlocks," in *Proc. of SOSP*, 2003, pp. 237–252.

[22] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," in *Proceedings of the 11th Australian Computer Science Conference*, vol. 10, no. 1, 1988, pp. 56–66.

[23] C. Flanagan and S. N. Freund, "FastTrack: Efficient and Precise Dynamic Race Detection," in *Proc. of PLDI*, 2009, pp. 121–133. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542490

[24] D. Hower, P. Dudnik, M. Hill, and D. Wood, "Calvin: Deterministic or Not? Free Will to Choose," in *Proc. of HPCA*, 2011, pp. 333 –334.

[25] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual: System Programming Guide, Part 1*, 2013.

[26] ISO JTC1/SC22/WG14, "ISO/IEC 9899:2011, Information Technology — Programming Languages — C," 2011.

[27] ISO JTC1/SC22/WG21, "ISO/IEC 14882:2011, Information Technology — Programming Languages — C++," 2011.

[28] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[29] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proc. of CGO*, 2004, pp. 75–86.

[30] N. G. Leveson and C. S. Turner, "An Investigation of the Therac-25 Accidents," *Computer*, vol. 26, no. 7, pp. 18–41, Jul. 1993. [Online]. Available: http://dx.doi.org/10.1109/MC.1993.274940

[31] T. Liu, C. Curtsinger, and E. D. Berger, "DTHREADS: Efficient Deterministic Multithreading," in *Proc. of SOSP*. ACM, 2011, pp. 327–336.

[32] K. Lu, X. Zhou, T. Bergan, and X. Wang, "Efficient Deterministic Multithreading Without Global Barriers," in *Proc. of PPOPP*, 2014, pp. 287–300.

[33] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm, "Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races," in *Proc. of ISCA*, 2010, pp. 210–221.

[34] A. Ludwin, V. Betz, and K. Padalia, "High-quality, Deterministic Parallel Placement for FPGAs on Commodity Hardware," in *Proc. of FPGA*, ser. FPGA '08. New York, NY, USA: ACM, 2008, pp. 14–23. [Online]. Available: http://doi.acm.org/10.1145/1344671.1344676

[35] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proc. of PLDI*, 2005, pp. 190–200.

[36] J. Manson, W. Pugh, and S. V. Adve, "The java memory model," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '05. New York, NY, USA: ACM, 2005, pp. 378–391.

[37] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy, "DRFX: A Simple and Efficient Memory Model for Concurrent Programming Languages," in *Proc. of PLDI*, 2010, pp. 351–362.

[38] H. S. Matar, I. Kuru, S. Tasiran, and R. Dementiev, "Accelerating Precise Race Detection Using Commercially-Available Hardware Transactional Memory Support," in *WODET*, 2014.

[39] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Parallel and Distributed Algorithms*, vol. 1, no. 23, pp. 215–226, 1989.

[40] T. Merrifield and J. Eriksson, "Conversion: Multi-version Concurrency Control for Main Memory Segments," in *Proc. of EuroSys*, 2013, pp. 127–139.

[41] S. L. Min and J.-D. Choi, "An Efficient Cache-based Access Anomaly Detection Scheme," in *Proc. of ASPLOS*, 1991, pp. 235–244.

[42] H. Mushtaq, Z. Al-Ars, and K. Bertels, "Detlock: Portable and efficient deterministic execution for shared memory multicore systems," *High Performance Computing, Networking Storage and Analysis, SC Companion:*, vol. 0, pp. 721–730, 2012.

[43] A. Muzahid, S. Qi, and J. Torrellas, "Vulcan: Hardware Support for Detecting Sequential Consistency Violations Dynamically," in *Proc. of MICRO*, 2012, pp. 363–375.

[44] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas, "SigRace: Signature-based Data Race Detection," in *Proc. of ISCA*, 2009, pp. 337–348.

[45] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: Efficient Deterministic Multithreading in Software," in *Proc. of ASPLOS*, 2009, pp. 97–108.

[46] ——, "Scaling Deterministic Multithreading," in *WODET*, 2011.

[47] J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy, "... And Region Serializability for All," in *HotPAR*. Berkeley, CA: USENIX, 2013.

[48] E. Pozniansky and A. Schuster, "Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs," in *Proc. of PPOPP*, 2003, pp. 179–190.

[49] M. Prvulovic, "CORD: Cost-effective (and nearly overhead-free) Order-Recording and Data race detection," in *Proc. of HPCA*, 2006, pp. 232–243.

[50] M. Prvulovic and J. Torrellas, "ReEnact: Using Thread-level Speculation Mechanisms to Debug Data Races in Multithreaded Codes," in *Proc. of ISCA*, 2003, pp. 110–121.

[51] X. Qian, J. Torrellas, B. Sahelices, and D. Qian, "Volition: Scalable and Precise Sequential Consistency Violation Detection," in *Proc. of ASPLOS*, 2013, pp. 535–548.

[52] M. Roth, M. J. Best, C. Mustard, and A. Fedorova, "Deconstructing the Overhead in Parallel Applications," in *Proc. of IISWC*, 2012, pp. 59–68.

[53] C. Segulja and T. S. Abdelrahman, "What is the Cost of Weak Determinism?" in *Proc. of PACT*, 2014, pp. 99–112.

[54] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data Race Detection in Practice," in *Proc. of WBIA*, 2009, pp. 62–71.

[55] A. Singh, D. Marino, S. Narayanasamy, T. Millstein, and M. Musuvathi, "Efficient Processor Support for DRFx, a Memory Model with Exceptions," in *Proc. of ASPLOS*, 2011, pp. 53–66.

[56] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proc. of ISCA*, 1995, pp. 24–36.

[57] W. Zhao, L. Moser, and P. Melliar-Smith, "Deterministic Scheduling for Multithreaded Replicas," in *Proc. of WORDS*, 2005, pp. 74–81.